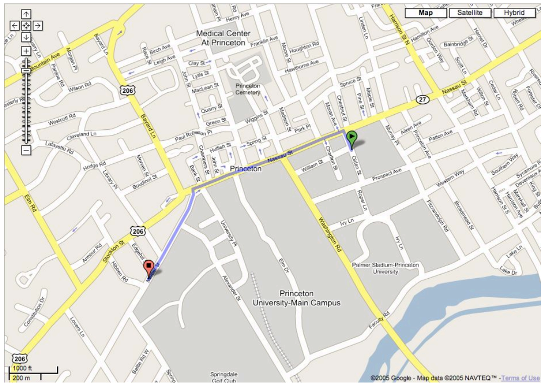
Shortest Paths

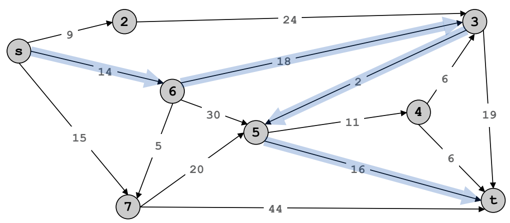


shortest path from Princeton CS department to Einstein's house

Shortest Path Problem

Shortest path problem. Given a weighted digraph, find the shortest directed path from s to t.

cost of path = sum of edge costs in path



Path: s→6→3→5→t
Cost: 14 + 18 + 2 + 16 = 50

Versions.

- Point-to-point, single source, all pairs.
- Nonnegative edge weights, arbitrary weights, Euclidean weights.

Brief History

Shimbel (1955). Information networks.

Ford (1956). RAND, economics of transportation.

Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, Seitz (1957).
Combat Development Dept. of the Army Electronic Proving Ground.

Dantzig (1958). Simplex method for linear programming.

Bellman (1958). Dynamic programming.

Moore (1959). Routing long-distance telephone calls for Bell Labs.

Dijkstra (1959). Simpler and faster version of Ford's algorithm.

Applications

More applications.

- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in higher level algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

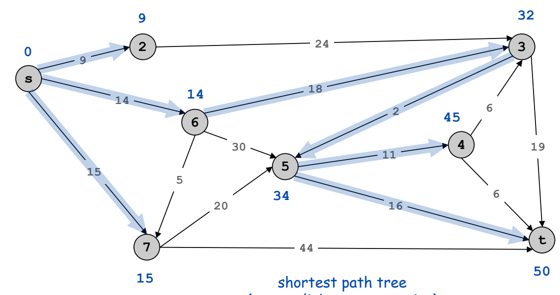
Reference: *Network Flows: Theory, Algorithms, and Applications*, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Dijkstra's Algorithm

Single Source Shortest Path

Assumptions.
- Digraph G.
- Single source `s`.
- Edge weights c(`v`, `w`) are nonnegative.

Goal. Find shortest path from `s` to every other vertex.



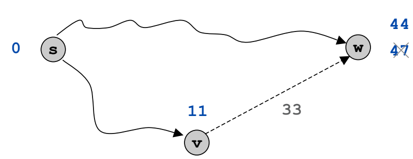
shortest path tree
(parent-link representation)

Edge Relaxation

Valid weights. For all vertices `v`, π(`v`) is length of some path from `s` to `v`.

Edge relaxation.
- Consider edge `e = v →w`.
- If current path from `s` to `v` plus edge `v→w` is shorter than current path to `w`, then update current path to `w`.

```
if (pi[w] > pi[v] + e.weight) {
    pi[w] = pi[v] + e.weight);
    pred[w] = v;
}
```



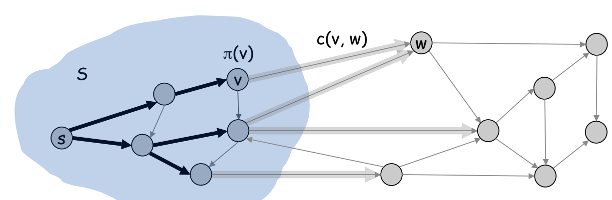
Dijkstra's Algorithm

Dijkstra's algorithm. Maintain set of weights π(`v`) and a set of explored vertices S for which π(`v`) is the length shortest `s`- `v` path.
- Initialize: S = { `s` }, π(`s`) = 0.
- Repeatedly choose unexplored node `w` which minimizes:

$$\pi(w) = \min_{(v,w)\,:\,v \in S} \pi(v) + c(v,w)$$

shortest path to some v in explored part, followed by a single edge e = (v, w)

  - set `pred[w] = v`
  - add `w` to S, and set π(`w`) = π(`v`) + `c(v, w)`



Dijkstra's Algorithm

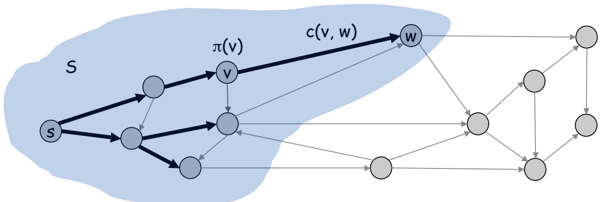
Dijkstra's algorithm. Maintain set of weights $\pi(v)$ and a set of explored vertices S for which $\pi(v)$ is the length shortest s- v path.

- Initialize: S = { s }, $\pi(s) = 0$.
- Repeatedly choose unexplored node w which minimizes:

$$\pi(w) = \min_{(v,w)\,:\,v \in S} \pi(v) + c(v,w)$$

shortest path to some v in explored part, followed by a single edge e = (v, w)

  - set pred[w] = v
  - add w to S, and set $\pi(w) = \pi(v) + c(v, w)$

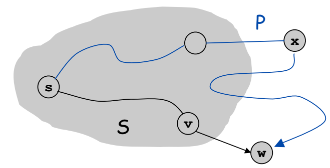


Dijkstra's Algorithm: Proof of Correctness

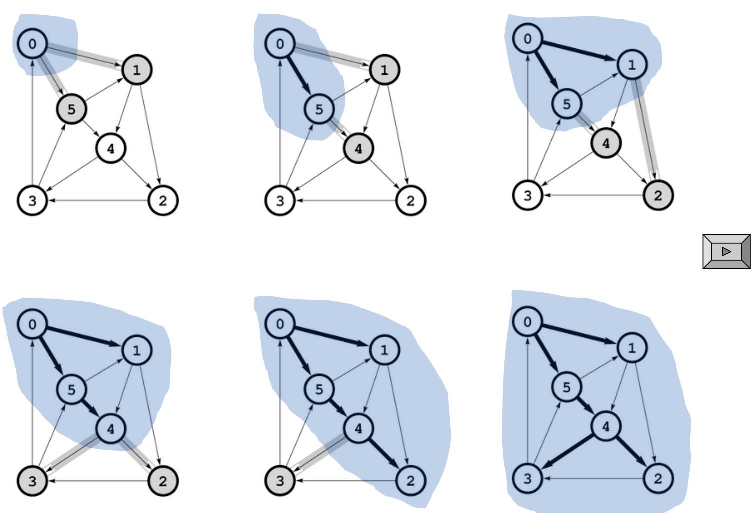
Invariant. For each vertex v in S, $\pi(v)$ is the length of shortest s-v path.

Pf. (by induction on |S|)

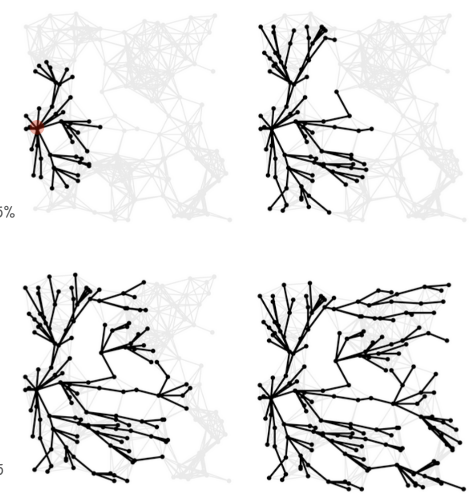
- Let w be next vertex added to S.
- $\pi(w) = \pi(v) + c(v, w)$ is length of some s-v path.
- Consider any s-v path P, and let x be first node on path outside S.
- P is already too long as soon as it reaches x by greedy choice.



Dijkstra's Algorithm



Shortest Path Tree



Dijkstra's Algorithm: Implementation

Critical step. Choose unexplored node w which minimizes:

$$\pi(w) = \min_{(v,w)\,:\,v \in S} \pi(v) + c(v, w)$$

Brute force implementation. Test all edges $\Rightarrow$ O(EV) time.

Dijkstra's Algorithm: Implementation

Critical step. Choose unexplored node w which minimizes:

$$\pi(w) = \min_{(v,w)\,:\,v \in S} \pi(v) + c(v, w)$$

Brute force implementation. Test all edges $\Rightarrow$ O(EV) time.

Efficient implementation. Maintain a priority queue of unexplored vertices, prioritized by π(w).

Q. How to maintain π?
A. When exploring `v`, for each edge `v->w` leaving `v`, update

$$\pi(w) = \min \{ \pi(w),\ \pi(v) + c(v, w) \}.$$

Weighted Edge

```
public class Edge {
   public final int source;
   public final int target;
   public final double weight;

   public Edge(int v, int w, double weight) {
      this.source = v;
      this.target = w;
      this.weight = weight;
   }

   public String toString() {
      return source + "->" + target + " (" + weight + ") ";
   }
}
```


Weighted Digraph

```
public class WeightedDigraph {
   private int V;
   private Sequence<Edge>[] adj;

   public WeightedDigraph(int V) {
      this.V = V;
      adj = (Sequence<Edge>[]) new Sequence[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Sequence<Edge>();
   }

   public int V() { return V; }

   public void addEdge(Edge e) { adj[e.source].add(e); }

   public Iterable<Edge> adj(int v) { return adj[v]; }

}
```

Dijkstra's Algorithm: Java Implementation

```java
public Dijkstra(WeightedDigraph G, int s) {
   pi = new double[G.V()];
   pred = new Edge[G.V()];
   for (int v = 0; v < G.V(); v++) pi[v] = INFINITY;

   IndexMinPQ<Double> pq = new IndexMinPQ<Double>(G.V());
   pi[s] = 0.0;
   pq.insert(s, pi[s]);

   while (!pq.isEmpty()) {
      int v = pq.delMin();
      for (Edge e : G.adj(v)) {
         int w = e.target;                                relax v-w
         if (pi[w] > pi[v] + e.weight) {
             pi[w] = pi[v] + e.weight;
             pred[w] = e;
             if (pq.contains(w)) pq.decrease(w, pi[w]);
             else                pq.insert(w, pi[w]);
         }
      }
   }
}
```


Indexed Priority Queue

Indexed PQ.
- Assume items are named `0` to `N-1`.
- Insert, delete min, test if empty. [PQ ops]
- Decrease key, contains. [ST-like ops]

`public class IndexMinPQ` (indexed priority queue)

	`IndexMinPQ(int N)`	create an empty PQ on N elements
`void`	`insert(int i, Key key)`	add element i with given key
`void`	`decrease(int i, Key key)`	decrease value of item i
`int`	`delMin()`	delete and return smallest item
`boolean`	`isEmpty()`	is the PQ empty?
`boolean`	`contains(int i)`	does the PQ contain item i?

Indexed Priority Queue: Array Implementation

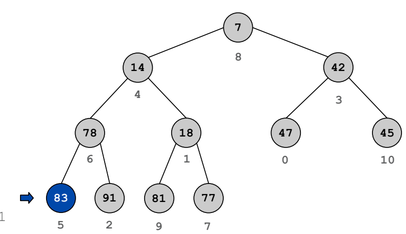
Indexed PQ: array implementation.
- Maintain vertex indexed array `keys[i]`.
- Insert key: change `keys[i]`.
- Decrease key: change `keys[i]`.
- Delete min: scan through `keys[i]` for each item `i`.
- Maintain a boolean array `marked[i]` to mark items in the PQ.

Operation	Array	Dijkstra
insert	1	× V
delete-min	V	× V
decrease-key	1	× E
is-empty	1	× V
contains	1	× V
total	V^2	

Indexed Priority Queue

Indexed PQ: binary heap implementation.
- Assume items are named `0` to `N-1`.
- Store priorities in a binary heap.

decrease key of element 5 from 83 to 31



i	pq	qp	key
0	-	6	47
1	8	5	18
2	4	9	91
3	3	3	42
4	6	2	14
5	1	8	83
6	0	4	78
7	10	11	77
8	5	1	7
9	2	10	81
10	9	7	45
11	7	-	-

How to decrease key of item i? Bubble it up.

How to know which heap node to bubble up? Maintains an extra array `qp[i]` that stores the heap index of item `i`.

Indexed Binary Heap: Java Implementation

```java
public class IndexMinPQ<Key extends Comparable> {
   private int N;
   private int[] pq, qp;
   private Comparable[] keys;

   public IndexMinPQ(int MAXN) {
      keys = new Comparable[MAXN + 1];
      pq = new int[MAXN + 1];
      qp = new int[MAXN + 1];
      for (int i = 0; i <= MAXN; i++) qp[i] = -1;
   }

   private boolean greater(int i, int j) {
      return keys[pq[i]].compareTo(keys[pq[j]]) > 0;
   }

   private void exch(int i, int j) {
      int swap = pq[i]; pq[i] = pq[j]; pq[j] = swap;
      qp[pq[i]] = i; qp[pq[j]] = j;
   }
```


Indexed Binary Heap: Java Implementation

```java
public void insert(int i, Key key) {
   N++;
   qp[i] = N;
   pq[N] = i;
   vals[i] = key;
   swim(N);
}

public int delMin() {
   int min = pq[1];
   qp[min] = -1;
   exch(1, N--);
   sink(1);
   return min;
}

public void decrease(int i, Key key) {
   keys[i] = key;
   swim(qp[i]);
}

public boolean contains(int i) { return qp[i] != -1; }
```


Dijkstra's Algorithm: Priority Queue Choice

The choice of priority queue matters in Dijkstra's implementation.

- Array: $\Theta(V^2)$.
- Binary heap: $O(E \log V)$.
- Fibonacci heap: $O(E + V \log V)$.

Operation	Array	Binary heap	Fib heap	Dijkstra
insert	1	log V	1 †	× V
delete-min	V	log V	log V †	× V
decrease-key	1	log V	1 †	× E
is-empty	1	1	1	× V
contains	1	1	1	× V
total	V^2	E log V	E + V log V	

† amortized

Dijkstra's Algorithm: Priority Queue Choice

The choice of priority queue matters in Dijkstra's implementation.

- Array: $\Theta(V^2)$.
- Binary heap: $O(E \log V)$.
- Fibonacci heap: $O(E + V \log V)$.

Best choice depends on sparsity of graph.

- 2,000 vertices, 1 million edges. Heap: 2-3x slower.
- 100,000 vertices, 1 million edges. Heap: 500x faster.
- 1 million vertices, 2 million edges. Heap: 10,000x faster.

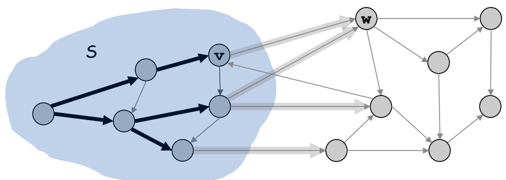
Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap far better for sparse graphs.
- Fibonacci heap best in theory, but not in practice.

Priority First Search

Priority first search. Maintain a set of explored vertices S, and grow S by exploring edges with exactly one endpoint leaving S.

DFS. Edge from vertex which was discovered most recently.
BFS. Edge from vertex which was discovered least recently.
Prim. Edge of minimum weight.
Dijkstra. Edge to vertex which is closest to `s`.



Edsger W. Dijkstra

> The question of whether computers can think is like the question of whether submarines can swim.
>
> Do only what only you can do.
>
> In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.
>
> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.
>
> APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

Edger Dijkstra
Turing award 1972

Bellman-Ford-Moore

Application: Currency Conversion

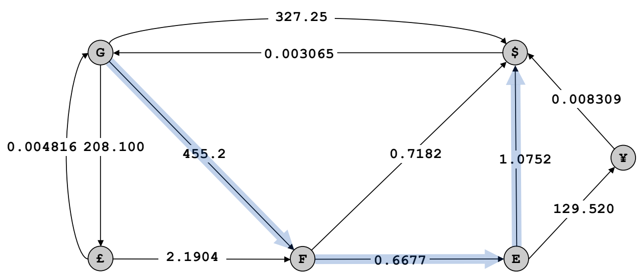
Currency conversion. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

- 1 oz. gold ⇒ $327.25.
- 1 oz. gold ⇒ £208.10 ⇒ ⇒ $327.00. [208.10 × 1.5714]
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28. [455.2 × .6677 × 1.0752]

Currency	£	Euro	¥	Franc	$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

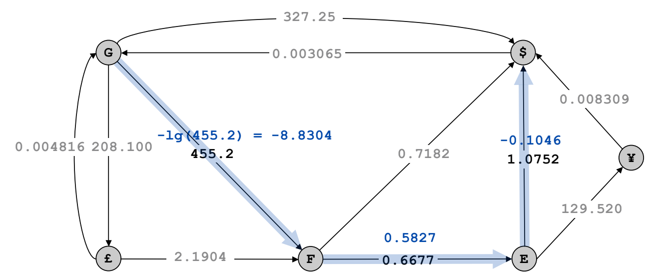
Application: Currency Conversion

Graph formulation.
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes product of weights.



Application: Currency Conversion

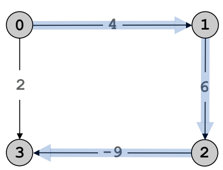
Reduction to shortest path problem.
- Let $\gamma(v, w)$ be exchange rate from currency v to w.
- Let $c(v, w) = - \lg \gamma(v, w)$.
- Shortest path with costs c corresponds to best exchange sequence.



Challenge. Solve shortest path problem with negative weights.

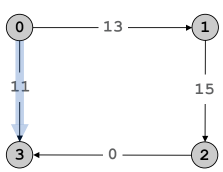
Shortest Paths with Negative Weights: Failed Attempts

Dijkstra. Can fail if negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

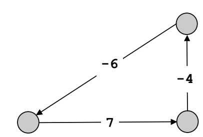
Re-weighting. Adding a constant to every edge weight can fail.



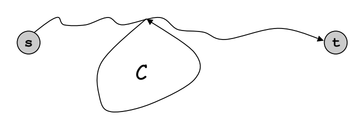
Adding 9 to each edge changes the shortest path.

Shortest Paths: Negative Cost Cycles

Negative cycle. Directed cycle whose sum of edge weights is negative.



Observation. If negative cycle C on path from `s` to `t`, then shortest path can be made arbitrarily negative by spinning around cycle; otherwise, there exists a shortest `s-t` path that is simple.



$\text{cost}(C) < 0$

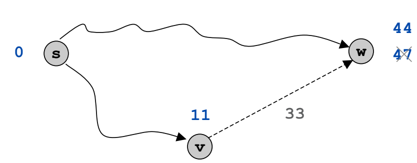
Review: Edge Relaxation

Valid weights. For all vertices `v`, `π(v)` is length of some path from `s` to `v`.

Edge relaxation.

- Consider edge `e = v →w`.
- If current path from `s` to `v` plus edge `v→w` is better than current path to `w`, then update current path to `w`.

```
if (pi[w] > pi[v] + e.weight) {
    pi[w] = pi[v] + e.weight);
    pred[w] = v;
}
```



Dynamic Programming

Dynamic programming algorithm.

- Initialize `pi[v] = ∞`, `pi[s]= 0`.
- Repeat `V` times: relax each edge `e`.

```
for (int i = 1; i <= V; i++) {    ← phase i
   for (int v = 0; v < G.V(); v++) {
      for (Edge e : G.adj(v)) {
         int w = e.target;                relax v-w
         if (pi[w] > pi[v] + e.weight) {
            pi[w] = pi[v] + e.weight)
            pred[w] = v;
         }
      }
   }
}
```


Dynamic Programming: Analysis

Running time. Θ(E V).

Invariant. At end of phase `i`, `pi[v]` ≤ length of any path from `s` to `v` using at most `i` edges.

Theorem. Assuming no negative cycles, upon termination `pi[v]` is the length of the shortest path from from `s` to `v`.

and `pred[v]` are the shortest paths

Bellman-Ford-Moore

Observation. If `pi[v]` doesn't change during phase `i`, no need to relax any edge leaving `v` in phase `i+1`.

FIFO implementation. Maintain queue of vertices whose distance changed.

be careful to keep at most one copy of each vertex on queue

Running time. Still Ω(E V) in worst case, but much faster in practice.

Bellman-Ford-Moore Algorithm

Initialize `pi[v]` = ∞ and `marked[v]= false` for all vertices `v`.

```
Queue<Integer> q = new Queue<Integer>();
marked[s] = true;
pi[s] = 0;
q.enqueue(s);

while (!q.isEmpty(v)) {
   int v = q.dequeue();
   marked[v] = false;
   for (Edge e : G.adj(v)) {
      int w = e.target;
      if (pi[w] > pi[v] + e.weight) {
          pi[w] = pi[v] + e.weight;
          pred[w] = e;
          if (!marked[w]) {
             marked[w] = true;
             q.enqueue(w);
          }
      }
   }
}
```


Single Source Shortest Paths Implementation: Cost Summary

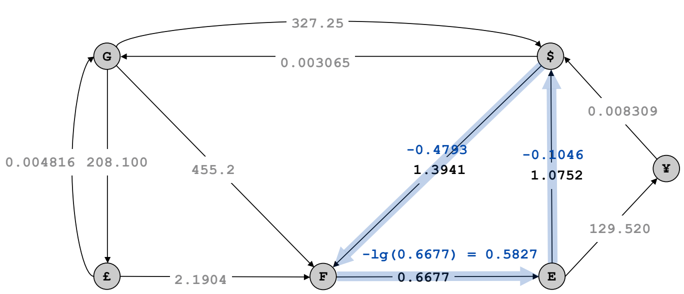
Algorithm	Worst Case	Best Case	Space
Dijkstra (classic) †	V^2	V^2	E
Dijkstra (heap) †	E log V	E	E
Dynamic programming ‡	E V	E V	E
Bellman-Ford ‡	E V	E	E

† nonnegative costs
‡ no negative cycles

Remark 1. Negative weights makes the problem harder.
Remark 2. Negative cycles makes the problem intractable.

Arbitrage

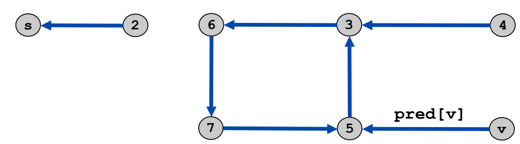
Arbitrage. Is there an arbitrage opportunity in currency graph?
- Ex: $1 ⇒ 1.3941 Francs ⇒ 0.9308 Euros ⇒ $1.00084.
- Is there a negative cost cycle?
- Fastest algorithm very valuable!



-0.4793 + 0.5827 - 0.1046 < 0

Negative Cycles

If negative cycle reachable from s. Bellman-Ford-Moore gets stuck in infinite loop, updating vertices in a cycle.

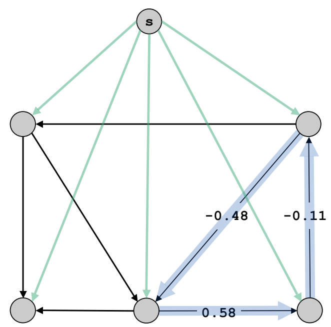


Finding a negative cycle. If any vertex `v` is updated in phase `V`, there exists a negative cycle, and we can trace back `pred[v]` to find it.

Negative Cycle Detection

Goal. Identify a negative cycle (reachable from any vertex).

Solution. Add 0-weight edge from artificial source `s` to each vertex `v`.
Run Bellman-Ford from vertex `s`.



Shortest Path in a DAG

Shortest/Longest Path in DAG

Shortest path in DAG algorithm.

- Consider vertices `v` in topological order:
  - relax each edge `v→w`

Theorem. Algorithm computes shortest path in linear time
(even if negative edge weights).